On the Generation of Initial Contexts for Effective Deadlock Detection (Extended Abstract)*

Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel

Complutense University of Madrid (UCM), Spain

Abstract. It has been recently proposed that testing based on symbolic execution can be used in conjunction with static deadlock analysis to define a deadlock detection framework that: (i) can show deadlock presence, in that case a concrete test-case and trace are obtained, and (ii) can also prove deadlock freedom. Such symbolic execution starts from an *initial distributed context*, i.e., a set of locations and their initial tasks. Considering all possibilities results in a combinatorial explosion on the different distributed contexts that must be considered. This paper proposes a technique to effectively generate initial contexts that can lead to deadlock, using the possible conflicting task interactions identified by static analysis, discarding other distributed contexts that cannot lead to deadlock. The proposed technique has been integrated in the above-mentioned deadlock detection framework hence enabling it to analyze systems without the need of any user supplied initial context.

1 Motivation

Deadlocks are one of the most common programming errors and they are therefore one of the main targets of verification and testing tools. We consider a distributed programming model with explicit *locations* (or distributed nodes) and *asynchronous* tasks that may be spawned and awaited among locations. Each location represents a processor with a procedure stack and an unordered queue of pending tasks. Initially all processors are idle. When an idle processor's task queue is non-empty, some task is selected for execution, this selection is non-deterministic. Let us see now our motivating example in Figure **??** which simulates a simple communication protocol between a database and a worker. Our implementation has the main method, and two classes Worker and DB implementing the worker and the database, respectively. The main method creates two distributed locations: the database and the worker, and (asynchronously) invokes methods register and work on each of them, respectively. The work method of a worker simply accesses the database (invoking asynchronously method getData) and then *blocks* until it gets the result, which is assigned to its data field. The instruction get blocks the execution in the current location until the awaited task has terminated. We use future variables [**?**,**?**] to detect the termination of

* This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006.

```
1 main(){
2     DB db = new DB();
3     Worker w = new Worker();
4     db!register(w);
5     w!work(db);}
6
7 class Worker{
8   Data data;
9   int work(DB db){
10    Fut⟨Data⟩ f = db!getData(this);
11    data = f.get;
12    return 0;
13  }
14  int ping(int n){return n;}
15 }// end of class Worker
16
17 class DB{
18  Data data = ...;
19  Worker client = null;
20  int connected = 1;
21  int makesConnection(){
22      connected = 3;
23      return connected;
24  }
25  int register(Worker w){
26      connected = 5;
27      Fut⟨int⟩ g = this!getData();
28      await g?;
29      if (connected > 0){
30          connected = connected − 1;
31          Fut⟨int⟩ f = w!ping(5);
32          if (f.get == 5) client = w;
33      }
34      return 0;
35  }
36  Data getData(Worker w){
37    if (client == w) return data;
38    else return null;
39  }
40 }// end of class DB
```

Fig. 1. Working example. Communication protocol between a DB and a worker

asynchronous tasks. The register method of the database makes a call to getData and waits for its execution. Once it has finished, it checks if the number of possible connections is bigger than 0. In that case connected is decreased by one, and the database makes sure that the worker is online. This is done by invoking asynchronously method ping with a concrete value and blocking until it gets the result with the same value. Then, the database registers the provided worker reference storing it in its client field. Method getData of the database returns its data field if the caller worker is registered, otherwise it returns **null**. Finally, method makesConnection sets the field connected to 3. Depending on the sequence of interleavings, the execution of this program can finish: (1) as one would expect, i.e., with worker.data = db.data, (2) with w.data = **null** if getData is executed before the assignment at line **??**, or, (3) in a deadlock.

We have recently proposed a deadlock detection framework [**?**,**?**] that combines static analysis and symbolic execution based testing [**?**,**?**,**?**,**?**]. The deadlock analysis (for instance, [**?**]) is first used to obtain descriptions of potential deadlock cycles which are then used to guide the testing process. The resulting deadlock detection framework hence can: (i) show deadlock presence, in which case a concrete test-case and trace are obtained, and (ii) prove deadlock freedom (up to the symbolic execution exploration limit). However, the symbolic execution phase needs to start from a concrete initial distributed context, i.e., a set of locations and their initial tasks. In our example, such an initial context is provided by the main method, which creates a Database and a Worker location,

and schedules a work task on the worker with the database as parameter, and, a register task on the database with the worker as parameter. This is however only one out of the possible contexts, and, of course, it could be the case that it does not expose an error that occurs in other contexts (for instance, it does not manifest any deadlock). This clearly limits the framework potential.

A fundamental challenge for a symbolic execution framework of distributed programs is to automatically and systematically generate *relevant* distributed contexts for the type of error that it aims at detecting. This would allow for instance applying symbolic execution for system and integration testing. The generation of relevant contexts involves two challenging aspects: (1) A first challenge is related to the elimination of redundant (useless) contexts. Observe that there is a combinatorial explosion on the different possible distributed contexts that can be generated when one considers all possible types and number of distributed locations and tasks within them. Therefore, it is crucial to provide the *minimal* set of initial contexts that contains only one representative of equivalent contexts. (2) For the particular type of error that one aims at detecting, an additional challenge is to be able to only generate initial contexts in which the error can occur. In the case of generating initial contexts for deadlock detection in our working example, this would mean generating for instance, a context with a database location and some worker location with a scheduled work task and a register task on the database for it, i.e., the context created by the main method. For instance, contexts that do not include both tasks would be useless for deadlock detection. Let us observe that if the assignment at Line **??** is changed to assign 0, then the initial contexts must also include a makesConnection task, otherwise no deadlock will be produced. Interestingly, deadlock analyses provide [**?**,**?**,**?**] potential *deadlock cycles* which contain the possibly conflicting task interactions that can lead to deadlock. This information will be used to help our framework anticipate this information and discard initial distributed contexts that cannot lead to deadlock from the beginning. Briefly, the main contributions of this work are twofold:

- We introduce the concept of *minimal* set of initial contexts and extend a static testing framework to automatically and systematically generate them.
- We present a deadlock-guided approach to effectively generate initial contexts for deadlock detection.

In an extended version of this work, we will validate experimentally our proposal and prove its soundness formally.

2 Asynchronous Programs

A program consists of a set of classes that define the types of locations, each of them defines a set of fields and methods of the form $M ::= T\ m(\bar{T}\ \bar{x})\{s\}$, where statements s take the form s::=s; s | x=e | **if** e then s **else** s | **while** e **do** s | **return** x; | b=**new** T($\bar{z}$) | f = x ! m($\bar{z}$) | **await** f? | x = f.**get**. Syntactically, a location will therefore be similar to a *concurrent object* that can be dynamically created

using the instruction **new** $\mathsf{T}(\bar{\mathsf{z}})$. The declaration of a future variable is as follows $\mathsf{Fut}\langle\mathsf{T}\rangle$ f, where T is the type of the result r, it adds a new future variable to the state. Instruction $\mathsf{f} = \mathsf{x}\ !\ \mathsf{m}(\bar{\mathsf{z}})$ spawns a new task (instance of method m) and it is set to the future f in the state. Instruction **await** f? allows non-blocking synchronization. If the future variable f we are awaiting for points to a finished task, then the **await** can be completed. Otherwise the task yields the lock so that any other task of the same location can take it. On the other hand, instruction f.**get** allows blocking synchronization. It waits for the future variable without yielding the lock, i.e., it blocks the execution of the location until the task that is awaiting is finished. Then, when the future is ready, it retrieves the result and allows continuing the execution. This instruction introduces possible deadlocks in the program, as two tasks can be awaiting for termination of tasks on each other's locations. Finally, instruction **return** x; releases the lock that will never be taken again by that task. Consequently, that task is *finished* and removed from the task queue. All statements of a task takes place serially (without interleaving with any other task) until it gets to a **return** or **await** f? instruction. Then, the processor becomes idle again, chooses non-deterministically the next pending task, and so on.

A *program state* or *configuration* is a set of locations $\{loc_0, ..., loc_n\}$. A *location* is a term $loc(o, tk, h, \mathcal{Q})$ where o is the location identifier, tk is the identifier of the *active task* that holds the location's lock or $\perp$ if the location's lock is free, h is its local heap, and $\mathcal{Q}$ is the set of tasks in the location. A *task* is a term $tsk(tk, m, l, s)$ where tk is a unique task identifier, m is the method name executing in the task, l is a mapping from local variables to their values, and s is the sequence of instructions to be executed. We assume that the execution starts from a *main* method without parameters. The initial state is S=$\{loc(0, 0, \perp, \{tsk(0, main, l, body(main))\}$ with an initial location with identifier 0 executing task 0, maps local variables to their initial values, and $body(m)$ is the sequence of instructions in method m and $ini(main)$ is the initial program point in method m. From now on, we represent the state as a Prolog list, and we write $[x \mapsto v]$ to denote $h(x) = v$ (resp. $l(x) = v$), that is, field x in the heap h (resp. local variable x in the mapping l) takes the value v.

In what follows, a *derivation* or *execution* [**?**] is a sequence of states $S_0 \xrightarrow{o_1.t_1} ... \xrightarrow{o_n.t_n} S_n$, where $S_i \xrightarrow{o_i.t_i} S_{i+1}$ denotes the execution of task t_i in location $o_i \in S_i$. The derivation is *complete* if S_0 is the initial state and $\nexists\ loc(o, \_, \_, \{tk\} \cup \mathcal{Q}) \in S_n$ such that $S_n \xrightarrow{o.tk} S_{n+1}$ and $S_n \neq S_{n+1}$. Given a state S, $exec(S)$ denotes the set of all possible complete executions starting at S.

3 Specifying and Generating Initial Contexts

In our asynchronous programs, the most *general* initial contexts consist of sets of locations with *free* variables in their fields, and initial tasks in each location queue with *free* variables as parameters, i.e., neither the fields nor the parameters have concrete values. A first approach to systematically generate initial

contexts could consist in generating, on backtracking, all possible multisets of initial tasks (method names), and for each one, generate all aliasing combinations with the locations of the tasks belonging to the same type of location. They are multisets because there can be multiple occurrences of the same task. To guarantee termination of this process we need to impose some limit in the generation of the multisets. For this, we could simply set a limit on the multiset global size. However it would be more reasonable and useful to set a limit on the maximum cardinality of each element in the multiset. To allow further flexibility, let us also set a limit on the minimum cardinality of each element. For instance, if we have a program with just one location type A with just one method m, and we set 1 and 2 as the minimum and maximum cardinalities respectively, then there are two possible multisets, namely, $\{m\}$ and $\{m, m\}$. The first one leads to one initial context with one location of type A with an instance of task m in its queue. The second one leads to two contexts, one with one location of type A with two instances of task m in its queue, and the other one with two different locations, each with an instance of task m in its queue.

On the other hand, it makes sense to allow specifying which tasks should be considered as initial tasks and which should not. A typical scenario is that the user knows which are the main tasks of the application and does not want to consider auxiliary or internal tasks as initial tasks. Another scenario is in the context of integration testing, where the tester might want to try out together different groups of tasks to observe how they interfere with each other. Also, the use of static analysis can help determine a subset of tasks of interest to detect some specific property. This is the case of our deadlock-guided approach of Section **??**. With all this, the input to our automatic generation of initial contexts is: (1) a set of *abstract tasks* $\mathcal{T}_{ini}$ such that each task is abstracted by the method name that is executing, (2) the minimum and maximum cardinalities. Thus, an initial context is a set of tuples ($\mathsf{C.M}$,C^{min}, C^{max}), where C and M are the class and method name resp., and C^{min} resp. C^{max} is the associated minimum resp. maximum cardinality. Note that this does not limit the approach in any way since one could just include in $\mathcal{T}_{ini}$ all methods in the program and set $C^{min} = 0$ and a sufficiently large C^{max}.

Example 1. Let us consider the set $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.getData}, 0, 1)\}$. The corresponding multisets are $\{\mathsf{register}\}$ and $\{\mathsf{register}, \mathsf{getData}\}$. All contexts must contain exactly one instance of task register and at most one instance of task getData. This leads to three possible contexts: (1) a DB location instance with a task register in its queue, (2) a DB location instance with tasks register and getData in its queue, and, (3) two different DB location instances, one of them with an instance of task register and the other one with an instance of task getData. For instance, the state corresponding to the latter context would be:

$$\begin{aligned}\mathtt{S} = [&\mathtt{loc}(\mathtt{DB1}, \mathtt{bot}, [\mathtt{data} \mapsto \mathtt{D1}, \mathtt{clients} \mapsto \mathtt{Cl1}, \mathtt{checkOn} \mapsto \mathtt{B1}],\\ &\qquad [\mathtt{tsk}(1, \mathtt{register}, [\mathtt{this} \mapsto \mathtt{r}(\mathtt{DB1}), \mathtt{m} \mapsto \mathtt{W1}], \mathtt{body}(\mathtt{register}))])\\ &\mathtt{loc}(\mathtt{DB2}, \mathtt{bot}, [\mathtt{data} \mapsto \mathtt{D2}, \mathtt{clients} \mapsto \mathtt{Cl2}, \mathtt{checkOn} \mapsto \mathtt{B2}],\\ &\qquad [\mathtt{tsk}(2, \mathtt{getData}, [\mathtt{this} \mapsto \mathtt{r}(\mathtt{DB2}), \mathtt{m} \mapsto \mathtt{W2}], \mathtt{body}(\mathtt{getData}))])],\end{aligned}$$

where D1, Cl1, and B1 (resp. D2, Cl2, and B2) are the fields data, clients, and checkOn of location DB1 (resp. DB2), and W1 resp. W2 the parameter of the task register resp. getData, and body(m) is the sequence of instructions in method m. Note that both fields and task parameters are fresh variables so that the context is the most general possible. Note that the first parameter of a task is always the location this and it is therefore fixed. □

In the following, we formally define the contexts that must be produced from a set of abstract tasks $\mathcal{T}_{ini}$ with associated cardinalities, and a procedure (as a Prolog rule) that generates these contexts as partially instantiated states. We use the notation $\{[m_1, ..., m_n]_{o_i}\}$ for an initial context where there exists a location $loc(o_i, \bot, h, \{tk(tk_1, m_1, l_1, body(m_1))\} \cup ... \cup \{tk(tk_n, m_n, l_n, body(m_n))\})$. Note that we can have $m_i = m_j$ with $i \neq j$. For instance, the three contexts in Example **??** are written as $\{[\mathsf{register}]_{\mathsf{db}_1}\}, \{[\mathsf{register}, \mathsf{getData}]_{\mathsf{db}_1}\}$ and $\{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{getData}]_{\mathsf{db}_2}\}$, respectively. Let us first define the set of initial contexts from a given $\mathcal{T}_{ini}$ when all tasks belong to the same class.

Definition 1 (Superset of initial contexts (same class C_i)). *Let $\mathcal{T}_{ini} = \{(C_i.m_1, C_1^{min}, C_1^{max}), \ldots, (C_i.m_n, C_n^{min}, C_n^{max})\}$ be the set of abstract tasks with associated cardinalities. Let us have $\sum_{i=1}^{n} C_i^{max}$ different identifiers: $o_{1,1}, \ldots, o_{1,C_1^{max}}$, $\ldots, o_{n,1}, \ldots, o_{n,C_n^{max}}$. We can find at most $\sum_{i=1}^{n} C_i^{max}$ instances of class C_i, that is, each abstract task m_i ($i \in [1, n]$) has at most C_i^{max} instances and each of them can be inside a different instance of class C_i. Let $u_{i,j}^{m_k}$ be an integer variable that denotes the number of instances of task m_k inside the location $o_{i,j}$ and let us consider the following integer system:*

$$\begin{cases} C_1^{min} \leq u_{1,1}^{m_1} + \ldots + u_{1,C_1^{max}}^{m_1} + \ldots + u_{n,1}^{m_1} + \ldots + u_{n,C_n^{max}}^{m_1} \leq C_1^{max} \\ \ldots \\ C_n^{min} \leq u_{1,1}^{m_n} + \ldots + u_{1,C_1^{max}}^{m_n} + \ldots + u_{n,1}^{m_n} + \ldots + u_{n,C_n^{max}}^{m_n} \leq C_n^{max} \end{cases}$$

Each formula requires at least C_k^{min} and at most C_k^{max} instances of task m_k. Each solution to this system corresponds to an initial context.
Let $(d_{1,1}^{m_1}, \ldots, d_{n,C_n^{max}}^{m_1}, \ldots, d_{1,1}^{m_n}, \ldots, d_{n,C_n^{max}}^{m_n})$ be a solution, then the corresponding initial context contains:

- *$loc(o_{i,j}, \bot, h, \mathcal{Q})$, that is, a location $o_{i,j}$ whose lock is free, the fields in h are mapped to fresh variables, and the queue $\mathcal{Q}$ contains: $d_{i,j}^{m_1}$ instances of abstract task m_1,..., and $d_{i,j}^{m_n}$ instances of m_n, if $i \in [1, n]$, $j \in [1, C_i^{max}]$ and $\exists d_{i,j}^{m_k} > 0, k \in [1, n]$, where each instance of m_i is $tsk(tk, m_i, l, body(m_i))$ and every argument in l is mapped to a fresh variable.*

Example 2. Let us consider the example $\mathcal{T}_{ini}$=$\{(\mathsf{DB.register}, 0, 1), (\mathsf{DB.getData}, 1, 1)\}$. The identifiers are $o_{1,1}$ and $o_{2,1}$, and the variables of the system are $u_{1,1}^{reg}$, $u_{2,1}^{reg}$, $u_{1,1}^{get}$ and $u_{2,1}^{get}$. Finally, we obtain the next system:

$$\begin{cases} 0 \leq u_{1,1}^{reg} + u_{2,1}^{reg} \leq 1 \\ 1 \leq u_{1,1}^{get} + u_{2,1}^{get} \leq 1 \end{cases}$$

We obtain 6 solutions: $(0,0,1,0), (0,0,0,1), (1,0,1,0), (1,0,0,1), (0,1,1,0)$ and $(0,1,0,1)$. Then, the superset of initial contexts is

$$\{\{[\mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{getData}]_{\mathsf{o}_{2,1}}\}, \{[\mathsf{register}, \mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}, \mathsf{getData}]_{\mathsf{o}_{2,1}}\},$$

$$\{[\mathsf{register}]_{\mathsf{o}_{2,1}}, [\mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}]_{\mathsf{o}_{1,1}}, [\mathsf{getData}]_{\mathsf{o}_{2,1}}\}\}$$

□

Let us observe that the two last contexts are equivalent since they are both composed of two instances of DB with tasks register and getData respectively. Therefore, we only need to consider one of these two contexts for symbolic execution. Considering both would lead to *redundancy*. The notion of minimal set of initial contexts below eliminates redundant contexts, hence avoiding useless executions.

Definition 2 (Equivalence relation $\sim$). *Two contexts C_1 and C_2 are equivalent, written $C_1 \sim C_2$, if $C_1 = C_2 = \emptyset$ or $C_1 = \{loc(o_1, \bot, h_1, \mathcal{Q}_1)\} \cup C_1'$, and $\exists\ o_2 \in C_2$ such that:*

1. *$C_2 = \{loc(o_2, \bot, h_2, \mathcal{Q}_2)\} \cup C_2'$,*
2. *$\mathcal{Q}_1$ and $\mathcal{Q}_2$ contain the same number of instances of each task, and*
3. *$C_1' \sim C_2'$.*

Example 3. The superset in Example **??** contains 3 equivalence classes induced by the relation $\sim$: (1) the class $\{\{[\mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{getData}]_{\mathsf{o}_{2,1}}\}\}$, where both contexts are composed of a location with a task getData, (2) the class $\{\{[\mathsf{register}, \mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}, \mathsf{getData}]_{\mathsf{o}_{2,1}}\}\}$, whose locations have two tasks register and getData. and, finally, (3) the class $\{\{[\mathsf{register}]_{\mathsf{o}_{2,1}}, [\mathsf{getData}]_{\mathsf{o}_{1,1}}\}, \{[\mathsf{register}]_{\mathsf{o}_{1,1}}, [\mathsf{getData}]_{\mathsf{o}_{2,1}}\}\}$, where both contexts have two locations with a task register and a task getData, respectively. □

Definition 3 (Minimal set of initial contexts $\mathcal{I}^{C_i}$ (same class Cl_i)). *Let $\mathcal{T}_{ini}$ be the set of abstract tasks, then the* minimal set of initial contexts *$\mathcal{I}^{Cl_i}$ is composed of a representative of each equivalence class induced by the relation $\sim$ over the superset of initial contexts for the input $\mathcal{T}_{ini}$.*

Example 4. As we have seen in the previous example, there are three different equivalence classes. So, the minimal set of initial contexts is composed of a representative of each class (we have renamed the identifiers for the sake of clarity):

$$\mathcal{I}^{DB} = \{\{[\mathsf{getData}]_{\mathsf{db}_1}\}, \{[\mathsf{register}, \mathsf{getData}]_{\mathsf{db}_1}\}, \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{getData}]_{\mathsf{db}_2}\}\}$$

□

Let us now define the set of initial contexts $\mathcal{I}$ when the input set $\mathcal{T}_{ini}$ contains tasks of different types of locations.

Definition 4 (Minimal set of initial contexts $\mathcal{I}$ (Different classes)). *Let $\mathcal{T}_{ini} = \{(C_1.m_1, C_1^{min}, C_1^{max}), \ldots, (C_n.m_n, C_n^{min}, C_n^{max})\}$ be the set of abstract tasks with associated cardinalities, and let us consider a partition of this set where every equivalence class is composed of abstract tasks of the same class. Hence, we have: $\mathcal{T}_{ini}^{C_1} = \{C_1.m'_1, \ldots, C_1.m'_{j_1}\}, \ldots, \mathcal{T}_{ini}^{C_n} = \{C_n.m''_1, \ldots, C_n.m''_{j_n}\}$ where $C_i \neq C_j, \forall i, j \in [1, n], i \neq j$.*
Then, let $\mathcal{I}^{C_i}$ be the minimal set of initial contexts for the input $\mathcal{T}_{ini}^{C_i}$, $i \in [1, n]$ and $U : \mathcal{I}^{C_1} \times \ldots \times \mathcal{I}^{C_n} \to \mathcal{I}$, defined by $U(s_1, \ldots, s_n) = s_1 \cup \ldots \cup s_n$. The set $\mathcal{I}$ is defined by the image set of application U.

Example 5. Let us consider the set $\mathcal{T}_{ini} = \{(\mathsf{DB.register}, 1, 1), (\mathsf{DB.getData}, 1, 1), (\mathsf{Worker.work}, 1, 1)\}$ from which we get the initial contexts $\mathcal{I}^{Worker} = \{\{[\mathsf{work}]_{\mathsf{w}_1}\}\}$ and $\mathcal{I}^{DB} = \{\{[\mathsf{register}, \mathsf{getData}]_{\mathsf{db},1}\}, \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{getData}]_{\mathsf{db}_2}\}\}$. Then, by Def. **??**,

$$\mathcal{I} = \{\{[\mathsf{register}, \mathsf{getData}]_{\mathsf{db}_1}, [\mathsf{work}]_{\mathsf{w}_1}\}, \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{getData}]_{\mathsf{db}_2}, [\mathsf{work}]_{\mathsf{w}_1}\}\}$$

□

We now define a Prolog predicate that generates the minimal set of initial contexts as partially instantiated states. Predicate `generate_contexts/2` in Figure **??** receives a set of abstract tasks with their associated maximum and minimum cardinalities, and generates on backtracking all generated initial contexts by means of `add_calls/3`. Predicate `normal_form/2` produces a normal form for the new context which is the same for all initial contexts in the same equivalence class. The new context is therefore only generated if it has not been previously generated (i.e., if the call `prev_generated/1` fails). The first rule of `add_calls/3` checks if the number of instances `Instances` of task `M` is smaller than the maximum cardinality `MaxC`, in which case we add a new instance of `M`, `Instances` is incremented, and, `add_calls/3` is recursively invoked. The second rule checks if the number of instances is greater than or equal to `Min`, it initializes the number of instances for the next method (`M`) and makes the recursive call to `add_calls/3`. Finally, the third rule corresponds to the base case when we are processing the last method of the list and the number of instances if greater than or equal to `Min`. Predicate `add_task/3` adds a new instance of method `M` to the current location. Note here that it can add the new task to one of the existing locations in `Locs` or create a new one to add it. The first rule checks if `SIn` is a variable (the end of the locations list) and then, it creates a new location to add the task `M`. To do so, we initialize the location fields, the method arguments and its tasks queue with a new task with fresh identifier `TkId`, and the instructions of method `M` (`body(M)`). The second rule checks if method `M` can be added to the first location by checking if the class of location `Id` matches with the class of `M`. If it does, then we add a new task to its tasks queue `Q`. The third rule ignores the first location and tries to add `M` to `SIn`.

Example 6. Let us show predicate `generate_contexts/2` in action for the set $\mathcal{T}_{ini} = \{(\mathsf{DB.reg}, 1, 1), (\mathsf{DB.make}, 1, 1), \mathsf{Worker.work}, 1, 1)\}$. The first rule of `add_calls/3` is applied, as `0 = Instances < MaxC = 1`. Then, `add_task/3` is

```
41 generate_contexts([(M,MinC,MaxC)|Methods],SOut) :-
42    add_calls([],[(M,0,MinC,MaxC)|Methods],SOut),
43    normal_form(SOut,N),
44    (prev_generated(N) -> fail ; assertz(prev_generated(N))).
45
46 add_calls(SIn,[(M,Instances,MinC,MaxC)|Ms],SOut) :-
47    Instances < MaxC,
48    add_task(SIn,M,SAux),
49    I2 is Instances + 1,
50    add_calls(SAux,[(M,I2,MinC,MaxC)|Methods],SOut).
51 add_calls(SIn,[(_,I,Min,_),(M,MinC,MaxC)|Methods],SOut) :-
52    Min <= I,
53    add_calls(SIn,[(M,0,MinC,MaxC)|Methods],SOut).
54 add_calls(SIn,[(_,I,Min,_)],SIn) :-
55    Min <= I.
56
57 add_task([],M,SIn) :-
58    fresh_location(LocId),fresh_task(TkId),
59    initialize_fields(M,Fields),initialize_mapping(M,L),
60    SIn = [loc(LocId,⊥,Fields,[tsk(TkId,M,L,body(M))])].
61 add_task([Loc|SIn],M,[Loc2|SIn]) :-
62    Loc = loc(Id,Lock,Fields,Q),
63    class(Id,Class), class(M,Class),
64    fresh_task(TkId), initialize_mapping(M,L),
65    Loc2 = loc(Id,Lock,Fields,[tsk(TkId,M,L,body(M))|Q]).
66 add_task([Loc|SIn],M,[Loc|SOut]) :-
67    add_task(SIn,M,SOut).
```

Fig. 2. Prolog predicate to generate minimal set of initial contexts

called with variable `Locs` and `M = DB.register`, at line **??**. As `Locs` is a variable, only the first rule of `add_task/3` can be applied and then, a new location is created (line **??**). Once this predicate has finished, `Instances` is incremented and `add_calls` is recursively called (line **??**). Now, the second rule is applied, as `0 = Min < Instances = 1`, and `add_calls` is called with `M = DB.makesConnection` whose number of instances is initialized to 0 (line **??**). Again, at line**??**, `add_call/3` is called with `M = DB.makesConnection` and `Locs` containing an instance of DB. Here we get to a branching point which gives rise to the two different initial contexts in Example **??**. In the first branch, `SIn` contains a location whose class is equal to that of the method `makesConnection`, so `LocVar` is the existing location and a new instance is added to its queue (lines **??** and **??**). Finally, `add_calls/3` is called with `M = Worker.work` (line **??**), it creates a new instance of class `Worker` with a task `work` (line **??**), it finishes correctly at line **??**, and returns an initial context containing an instance of DB with tasks register and makesConnection, and an instance of Worker with task work. Now,

it fails and the backtracking goes back to the branching point. Here, the third rule is applied and then, the first location is ignored and task `makesConnection` is added to a new location at line **??**. It finishes in a similar way. In this case, the initial context returned contains two instances of DB containing a task register and makesConnection, respectively, and an instance of Worker with task work. □

4 On Automatically Inferring Deadlock-Interfering Tasks

The systematic generation of initial contexts produces a combinatorial explosion and therefore it should be used with small sets of abstract tasks (and low cardinalities). However, in the context of deadlock detection, in order not to miss any deadlock situation, one has to consider in principle all methods in the program, hence producing scalability problems. Interestingly, it can happen that many of the tasks in the generated initial contexts do not affect in any way deadlock executions. Our challenge is to only generate initial contexts from which a deadlock can show up. For this, the deadlock analysis provides the possibly conflicting task interactions that can lead to deadlock. We propose to use this information to help our framework discard initial contexts that cannot lead to deadlock from the beginning. Section **??** summarizes the concepts of the deadlock analysis used to obtain the deadlock cycles, and Section **??** presents the algorithm to generate the set of initial tasks $\mathcal{T}_{ini}$.

4.1 Deadlock Analysis and Abstract Deadlock Cycles

The deadlock analysis of [**?**] returns a set of abstract deadlock cycles of the form $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1$, where $p_1, \ldots, p_n$ are program points, $tk_1, \ldots, tk_n$ are *task abstractions*, and nodes $e_1, \ldots, e_n$ are either *location abstractions* or task abstractions. The abstractions for tasks and locations can be performed at different levels of accuracy during the analysis: the simple abstraction that we will use for our formalization abstracts each concrete location o by the program point at which it is created o_{pp}, and each task by the method name executing (as in Section **??**). They are abstractions since there could be many locations created at the same program point and many tasks executing the same method. Points-to analysis [**?**,**?**] can be used to infer such abstractions with more precision, for instance, by distinguishing the actions performed by different location abstractions. Each arrow $e \xrightarrow{p:tk} e'$ should be interpreted like "abstract location or task e is waiting for the termination of abstract location or task e' due to the synchronization instruction at program point p of abstract task tk". Three kinds of arrows can be distinguished, namely, *task-task* (an abstract task is awaiting for the termination of another one), *task-location* (an abstract task is awaiting for an abstract location to be idle) and *location-task* (the abstract location is blocked due the abstract task). *Location-location* arrows cannot happen.

Example 7. In our working example there are two abstract locations, $o_{??}$, corresponding to location database created at line **??** and $o_{??}$, corresponding to the n locations worker, created inside the loop at line **??**; and four abstract tasks, *register*, *getD*, *work* and *ping*. The following cycle is inferred by the deadlock analysis: $o_{??} \xrightarrow{??:\mathsf{register}} \mathsf{ping} \xrightarrow{??:\mathsf{ping}} o_{??} \xrightarrow{??:\mathsf{work}} \mathsf{getD} \xrightarrow{??:\mathsf{getD}} o_{??}$. The first arrow captures that the location created at Line **??** is blocked waiting for the termination of task ping because of the synchronization at L**??** of task register. Also, a dependency between a task and a location (for instance, ping and $o_{??}$) captures that the task is trying to execute on that (possibly) blocked location. Abstract deadlock cycles can be provided by the analyzer to the user. But, as it can observed, it is complex to figure out from them why these dependencies arise, and more importantly the interleavings scheduled to lead to this situation. □

4.2 Generation of initial tasks

The underlying idea is as follows: we select an abstract cycle detected by the deadlock analysis, and extract a set of potential abstract tasks which can be involved in a deadlock. In a naive approximation, we could take those abstract tasks that are inside the cycle and contain a blocking instruction. We also need to set the maximum cardinality for each task to ensure finiteness (by default 1) and require at least one instance for each task (minimum cardinality).

This approach is valid as long as we only have blocking synchronization primitives, i.e., when the location state stays unchanged until the resumption of a suspended execution. However, this kind of concurrent/distributed languages usually include some sort of non-blocking synchronization primitive. When a location stops its execution due to an **await** instruction, another task can interleave its execution with it, i.e., start to execute and, thus, modify the location state (i.e., the location *fields*). Then, if a call or a blocking instruction involved in a deadlock depends on the value of one of these fields, and we do not consider all the possible values, a deadlock could be missed. As a consequence, we need to consider at release points, all possible interleavings with tasks that modify the fields in order to capture all deadlocks.

Let us consider now a simple modification of our working example. Line **??** is replaced by connected = 0. Now it is easy to see that if we only consider register and work as input, deadlocks are lost: once register is executed and the instruction at line **??** is reached, the location's queue only contains task getData but no makesConnection and, therefore, when task register is resumed, field connected stays unchanged and the body of the condition is not executed, so we cannot have a deadlock situation.

In the following we define the *deadlock-interfering* tasks for a given abstract deadlock cycle, i.e., an *over-approximation* of the set of tasks that need to be considered in initial contexts so that we cannot miss a representative of the given deadlock cycle. In our extended example, those would be, register and work but also makesConnection.

Definition 5 (initialTasks(C)). *Let C an abstract deadlock cycle. Then,*

$$initialTasks(C) := \bigcup_{i_{call} \in t \in C} initialTasks(t, i_{call}, C) \cup \bigcup_{i_{sync} \in t \in C} initialTasks(t, i_{sync}, C)$$

where:

- $initialTasks(t, i, C) = \emptyset \quad if\ o \xrightarrow{t} t_2 \notin C\ and\ i \neq i_{mod}\ and\ \nexists\ i_{await} \in [t_0, i]$
- $initialTasks(t, i, C) = \{t\}\ if\ (o \xrightarrow{t} t_2 \in C\ or\ i = i_{mod}\ )\ and\ \nexists\ i_{await} \in [t_0, i]$
- $initialTasks(t, i, C) = \{t\} \cup \bigcup_{f \in fields(i)} \left(\bigcup_{i_{mod} \in t_{mod} \in mods(f)} initialTasks(t_{mod}, i_{mod}, C) \right)$ $if\ \exists\ i_{await}\ \in [t_0, i]$

The definition relies on function fields(I) which, given an instruction I, returns the set of class fields that have been read or written until the execution of instruction I. Let mods(f) be the set of instructions that modify field f. We can observe that *initialTasks(C)* is the union of initial tasks for each relevant instruction inside the cycle C, i.e., asynchronous calls and synchronization primitives. We can also observe in the auxiliary function *initialTasks(t,i,C)* that: (1) if the instruction i is not producing a *location-task edge* and it is not an instruction modifying a field, then t does not need to be added as initial task, (2) if i produces a *location-task edge* or is modifying a field, and we do not have any **await** instruction between the beginning of the task and i, then i is going to be executed under the most general context, so we do not need to add more initial tasks but t, and (3) on the other hand, if there exists an **await** instruction between the beginning of task t, namely t_0, and instruction i, each field f inside the set fields(i) could be changed before the resumption of the **await** by any task modifying f. Thus, tasks containing any of the possible f-modifying instructions must be considered and, recursively, their initial tasks.

It is important to highlight that this definition could be infinite depending on the program we are working with. For instance, if we apply the definition to the abstract cycle C in Example **??**, $initialTasks$(db.register, **??**, C) will be evaluated. It fits well with the conditions on third clause, as there exists an **await** instruction, fields(**??**) = {connected} and then again **??** is a modifier instruction of field connected, so $initialTasks$(db.register, **??**, C) will be evaluated again recursively.

Figure **??** presents predicate `calculate_interfering_tasks/2` that finitely infers the interfering-tasks for a given deadlock cycle as defined by Def **??**. First, both the list of events and of answers are initialized (`init/5`) according to the type of edge. For each edge in the cycle, we take the call and the corresponding synchronization instruction, and we add them to the pending events. Moreover, **get** instructions produce *location-task* edges, so they are also included in the answers list, as they have to be inside the initial context. The other tasks included in the initial context are the ones which could affect the conditions of those instructions. In predicate `process_events/3`, we take a pending event `(Task,Inst)` and we check if there is an **await** instruction between the start of `Task` and `Inst`, using predicate (`thereis_await/2`), where the previously

```
68 calculate_interfering_tasks(Cycle,Tasks) :-
69    init(Cycle,[],Events,[],Ans),
70    process_events(Events,Ans,NoCardinality),
71    findall((Task,1,1),member((Task,_),NoCardinality),Repeated),
72    list_to_set(Repeated,Tasks).
73
74 init([],Evs,Evs,Ans,Ans).
75 init([edge(loc,get(Task,LAsync,LGet),task)|C],Evs,Evs2,Ans,Ans2) :-
76    !, init(C,[(Task,LAsync),(Task,LGet)|Evs],Evs2,[(Task,LGet)|Ans],Ans2).
77 init([edge(task,sync(Task,LAsync,LSync),task)|C],Evs,Evs2,Ans,Ans2) :-
78    !, init(C,[(Task,LAsync),(Task,LSync)|Evs],Evs2,Ans,Ans2).
79 init([_|Cycle],Evs,Evs2,Ans,Ans2) :- init(Cycle,Evs,Evs2,Ans,Ans2).
80
81 process_events([],Ans,Ans).
82 process_events([(Task,Inst)|Evs],Ans,Ans2) :-
83    thereis_await(Task,Inst),
84    accessed_fields(Task,Inst,Fields), !,
85    findall((T,L),(member(F,Fields),
86                   inst(F,write,T,L),
87                   \+ member((T,L),Ans)),Modifiers),
88    append(Modifiers,Evs,Evs2), append(Modifiers,Ans,Ans1),
89    process_events(Evs2,Ans1,Ans2).
90 process_events([_|Evs],Ans,Ans2) :- process_events(Evs,Ans,Ans2).
```

Fig. 3. Prolog predicate to infer interfering tasks for a given deadlock cycle

accessed field values (`accessed_fields/3`) could be changed (third clause in Def. **??**. In case it does, we need to include in the answer set all tasks which contain instructions modifying such field (`inst/4`). Besides, this change could be inside an if-else body and we also need to consider the fields inside such condition. Therefore we add the modifier instructions to the pending events list. This predicate finishes when this list is empty and `Ans` is the list of pairs with all interfering instructions and their container tasks. Finally, we only take the tasks, i.e., the first component of each pair, we set their minimum and maximum cardinalities and remove duplicates (`list_to_set/2`). Finiteness is guaranteed because each instruction is added to the pending events and answers lists at most once, and the number of instructions is finite.

Example 8. Let us show how predicate `calculate_interfering_tasks/2` works for our modified example. For the sake of clarity, instructions are identified by their line numbers. After the `init/5` predicate, the value of variables `Events` and `Ans` is [(Worker.work, **??**), (Worker.work, **??**), (DB.register, **??**), (DB.register, **??**)] and [(DB.register, **??**), (Worker.work, **??**)], respectively. Hence, predicate `process_events/3` takes (Worker.work, **??**) first. Since there is not an **await** instruction between the beginning of work and line **??**, `Ans` stays unchanged. The same happens with (Worker.work, **??**). Now, the pending events list is [(DB.register, **??**),

(DB.register, ??,)] and (DB.register, ??) is processed. Now, there is an **await** between lines ?? and ?? and, then, `fields(DB.register,??,Fields)` is invoked and `Fields` = [connected]. We find three instructions modifying the field connected: ?? ∈ DB.makesConnection, ?? ∈ DB.register and ?? ∈ DB.register. None of them is a member of the answer set and hence they are added to both lists. Now, `Evs` is [(DB.register,??), (DB.makesConnection,??), (DB.register,??),(DB.register,??)] but again there is no **await** between the beginning of tasks DB.register and DB.makesConnection and lines ?? and ??, respectively and, thus, `Ans` stays unchanged. Finally, both (DB.register, ??) and (DB.register, ??) are taken and both `fields(DB.register,??,Fields)` and `fields(DB.register,??,Fields)` hold where `Fields`=[connected], but the modifier instructions have been previously added to `Ans`, hence `Ans` remains unchanged, and the pending events list becomes empty. Finally, the algorithm projects over the first component of each pair in the list, sets the minimum and maximum cardinalities to 1 and removes duplicates, returning the set $\mathcal{T}_{ini}$ = {(DB.register,1,1), (Worker.work,1,1), (DB.makesConnection,1,1)}. Thus, the generation of initial contexts for this set (see Example ??) produces

$$\mathcal{I} = \{\ \{[\mathsf{register}, \mathsf{makesConnection}]_{\mathsf{db}_1}[\mathsf{work}]_{\mathsf{w}_1}\},\\ \{[\mathsf{register}]_{\mathsf{db}_1}, [\mathsf{makesConnection}]_{\mathsf{db}_2}, [\mathsf{work}]_{\mathsf{w}_1}\}\}$$

□

5 Conclusions and Related Work

We have proposed a framework for the automatic generation of initial contexts for deadlock-guided symbolic execution. Such initial contexts are composed of the interfering tasks which, according to a static deadlock analyzer, might lead to deadlock. Given the initial contexts, we can drive symbolic execution towards paths that are more likely to manifest a deadlock, discarding safe contexts. There is a large body of work on deadlock detection including both dynamic and static approaches. Much of the existing work, both for asynchronous programs [?,?] and thread-based programs [?,?], is based on static analysis techniques. Although we have used the static analysis of [?], the information provided by other deadlock analyzers could be used in an analogous way. Deadlock detection has been also studied in the context of dynamic testing and model checking [?,?,?], where sometimes has been combined with static information [?,?]. The initial contexts generated by our framework are of interest also in these approaches. Deadlock detection is even more challenging in the context of thread-based concurrency model. As future work, we plan to investigate how our framework could be adapted to this model.